2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems

Jishen Zhao*♮†, Onur Mutlu†, Yuan Xie‡*
*Pennsylvania State University, †Carnegie Mellon University, ‡University of California, Santa Barbara, ♮Hewlett-Packard Labs
*juz138@cse.psu.edu, †onur@cmu.edu, ‡yuanxie@ece.ucsb.edu

***Abstract*—Byte-addressable nonvolatile memories promise a new technology, persistent memory, which incorporates desirable attributes from both traditional main memory (byte-addressability and fast interface) and traditional storage (data persistence). To support data persistence, a persistent memory system requires sophisticated data duplication and ordering control for write requests. As a result, applications that manipulate persistent memory (persistent applications) have very different memory access characteristics than traditional (non-persistent) applications, as shown in this paper. Persistent applications introduce heavy write traffic to contiguous memory regions at a memory channel, which cannot concurrently service read and write requests, leading to memory bandwidth underutilization due to low bank-level parallelism, frequent write queue drains, and frequent bus turnarounds between reads and writes. These characteristics undermine the high-performance and fairness offered by conventional memory scheduling schemes designed for non-persistent applications.**

Our goal in this paper is to design a fair and high-performance memory control scheme for a persistent memory based system that runs both persistent and non-persistent applications. Our proposal, FIRM, consists of three key ideas. First, FIRM categorizes request sources as non-intensive, streaming, random and *persistent*, and forms batches of requests for each source. Second, FIRM strides persistent memory updates across multiple banks, thereby improving bank-level parallelism and hence memory bandwidth utilization of persistent memory accesses. Third, FIRM schedules read and write request batches from different sources in a manner that minimizes bus turnarounds and write queue drains. Our detailed evaluations show that, compared to five previous memory scheduler designs, FIRM provides significantly higher system performance and fairness.

***Index Terms*—memory scheduling; persistent memory; fairness; memory interference; nonvolatile memory; data persistence**

1. Introduction

For decades, computer systems have adopted a two-level storage model consisting of: 1) a fast, byte-addressable main memory that temporarily stores applications' working sets, which is lost on a system halt/reboot/crash, and 2) a slow, block-addressable storage device that permanently stores persistent data, which can survive across system boots/crashes. Recently, this traditional storage model is enriched by the new **persistent memory** technology – a new tier between traditional main memory and storage with attributes from both [2, 9, 47, 54, 59]. Persistent memory allows applications to perform loads and stores to manipulate persistent data, as if they are accessing traditional main memory. Yet, persistent memory is the permanent home of persistent data, which is protected by versioning (e.g., logging and shadow updates) [17, 54, 88, 90] and write-order control [17, 54, 66], borrowed from databases and file systems to provide consistency of data, as if data is stored in traditional storage devices (i.e., hard disks or flash memory). By enabling data persistence in main memory, applications can directly access persistent data through a fast memory interface without paging data blocks in and out of slow storage devices or performing context switches for page faults. As such, persistent memory can dramatically boost the performance of applications that require high reliability demand, such as *databases* and *file systems*, and enable the design of more robust systems at high performance. As a result, persistent memory has recently drawn significant interest from both academia and industry [1, 2, 16, 17, 37, 54, 60, 66, 70, 71, 88, 90]. Recent works [54, 92] even demonstrated a persistent memory system with performance close to that of a system without persistence support in memory.

Various types of physical devices can be used to build persistent memory, as long as they appear *byte-addressable* and *nonvolatile* to applications. Examples of such byte-addressable nonvolatile memories (BA-NVMs) include spin-transfer torque RAM (STT-MRAM) [31, 93], phase-change memory (PCM) [75, 81], resistive random-access memory (ReRAM) [14, 21], battery-backed DRAM [13, 18, 28], and nonvolatile dual in-line memory modules (NV-DIMMs) [89].[1]

As it is in its early stages of development, persistent memory especially serves applications that can benefit from reducing storage (or, persistent data) access latency with relatively few or lightweight changes to application programs, system software, and hardware [10]. Such applications include databases [90], file systems [1, 17], key-value stores [16], and persistent file caches [8, 10]. Other types of applications may not directly benefit from persistent memory, but can still use BA-NVMs as their working memory (nonvolatile main memory without persistence) to leverage the benefits of large capacity and low stand-by power [45, 73]. For example, a large number of recent works aim to fit BA-NVMs as part of main memory in the traditional two-level storage model [22, 23, 33, 45, 46, 57, 72, 73, 74, 91, 94]. Several very recent works [38, 52, 59] envision that BA-NVMs can be simultaneously used as persistent memory and working memory. In this paper, we call applications leveraging BA-NVMs to manipulate persistent data as **persistent applications**, and those using BA-NVMs solely as working memory as **non-persistent applications**.[2]

Most prior work focused on designing memory systems to accommodate either type of applications, persistent or non-persistent. Strikingly little attention has been paid to study the cases when these two types of applications *concurrently* run in a system. Persistent applications require the memory system to support *crash consistency*, or the *persistence property*, typically supported in traditional storage systems. This property guarantees that the system's data will be in a consistent state after a system or application crash, by ensuring that persistent memory updates are done carefully such that incomplete updates are recoverable. Doing so requires data duplication and careful control over the ordering of writes arriving at memory (Section 2.2). The sophisticated designs to support persistence lead to new memory access characteristics for persistent applications. In particular, we find that these applications have very high write intensity and very low memory bank parallelism due to *frequent streaming writes* to persistent data in memory (Section 3.1). These characteristics lead to substantial resource contention between reads and writes at the shared memory interface for a system that concurrently runs persistent

[1] STT-MRAM, PCM, and ReRAM are collectively called nonvolatile random-access memories (NVRAMs) or storage-class memories (SCMs) in recent studies [1, 52, 90]

[2] A system with BA-NVMs may also employ volatile DRAM, controlled by a separate memory controller [22, 57, 73, 74, 91]. As we show in this paper, significant resource contention exists at the BA-NVM memory interface of persistent memory systems between persistent and non-persistent applications. We do not focus on the DRAM interface.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.47

and non-persistent applications, unfairly slowing down either or both types of applications. Previous memory scheduling schemes, designed solely for non-persistent applications, become inefficient and low-performance under this new scenario (Section 3.2). We find that this is because the heavy write intensity and low bank parallelism of persistent applications lead to three key problems not handled well by past schemes: 1) frequent write queue drains in the memory controller, 2) frequent bus turnarounds between reads and writes, both of which lead to wasted cycles on the memory bus, and 3) low memory bandwidth utilization during writes to memory due to low memory bank parallelism, which leads to long periods during which memory reads are delayed (Section 3).

Our goal is to design a memory control scheme that achieves both fair memory access and high system throughput in a system concurrently running persistent and non-persistent applications. We propose FIRM, a fair and high-performance memory control scheme, which 1) improves the bandwidth utilization of persistent applications and 2) balances the bandwidth usage between persistent and non-persistent applications. FIRM achieves this using three components. First, it categorizes memory request sources as non-intensive, streaming, random and *persistent*, to ensure fair treatment across different sources, and forms batches of requests for each source in a manner that preserves row buffer locality. Second, FIRM strides *persistent* memory updates across multiple banks, thereby improving bank-level parallelism and hence memory bandwidth utilization of persistent memory accesses. Third, FIRM schedules read and write request batches from different sources in a manner that minimizes bus turnarounds and write queue drains. Compared to five previous memory scheduler designs, FIRM provides significantly higher system performance and fairness.

This paper makes the following contributions:

- We identify new problems related to resource contention at the shared memory interface *when persistent and non-persistent applications concurrently access memory*. The key fundamental problems, caused by memory access characteristics of persistent applications, are: 1) frequent write queue drains, 2) frequent bus turnarounds, both due to high memory write intensity, and 3) memory bandwidth underutilization due to low memory write parallelism. We describe the ineffectiveness of prior memory scheduling designs in handling these problems. (Section 3)
- We propose a new strided writing mechanism to improve the bank-level parallelism of persistent memory updates. This technique improves memory bandwidth utilization of memory writes and reduces the stall time of non-persistent applications' read requests. (Section 4.3)
- We propose a new *persistence-aware* memory scheduling policy between read and write requests of persistent and non-persistent applications to minimize memory interference and reduce unfair application slowdowns. This technique reduces the overhead of switching the memory bus between reads and writes by reducing bus turnarounds and write queue drains. (Section 4.4)
- We comprehensively compare the performance and fairness of our proposed persistent memory control mechanism, FIRM, to five prior memory schedulers across a variety of workloads and system configurations. Our results show that 1) FIRM provides the highest system performance and fairness on average and for all evaluated workloads, 2) FIRM's benefits are robust across system configurations, 3) FIRM minimizes the bus turnaround overhead present in prior scheduler designs. (Section 7)

2. Background

In this section, we provide background on existing memory scheduling schemes, the principles and mechanics of persistent memory, and the memory requests generated by persistent applications.

2.1. Conventional Memory Scheduling Mechanisms

A memory controller employs memory request buffers, physically or logically separated into a read and a write queue, to store the memory requests waiting to be scheduled for service. It also utilizes a memory scheduler to decide which memory request should be scheduled next. A large body of previous work developed various memory scheduling policies [7, 26, 27, 34, 41, 42, 48, 49, 61, 62, 63, 64, 65, 67, 76, 77, 84, 85, 95]. Traditional commodity systems employ a variant of the first-ready first-come-first-serve (FR-FCFS) scheduling policy [76, 77, 95], which prioritizes memory requests that are row-buffer hits over others and, after that, older memory requests over others. Because of this, it can unfairly deprioritize applications that have low buffer hit rate and that are not memory intensive, hurting both fairness and overall system throughput [61, 64]. Several designs [41, 42, 63, 64, 65, 67, 84, 85] aim to improve either system performance or fairness, or both. PAR-BS [65] provides fairness and starvation freedom by batching requests from different applications based on their arrival times and prioritizing the oldest batch over others. It also improves system throughput by preserving the bank-level parallelism of each application via the use of rank-based scheduling of applications. ATLAS [41] improves system throughput by prioritizing applications that have received the least memory service. However, it may unfairly deprioritize and slow down memory-intensive applications due to the strict ranking it employs between memory-intensive applications [41, 42]. To address this issue, TCM [42] dynamically classifies applications into two clusters, low and high memory-intensity, and employs heterogeneous scheduling policies across the clusters to optimize for both system throughput and fairness. TCM prioritizes the applications in the low-memory-intensity cluster over others, improving system throughput, and shuffles thread ranking between applications in the high-memory-intensity cluster, improving fairness and system throughput. While shown to be effective in a system that executes only non-persistent applications, unfortunately, none of these scheduling schemes address the memory request scheduling challenges posed by *concurrently-running persistent and non-persistent applications*, as we discuss in Section 3 and evaluate in detail in Section 7.[3]

2.2. Persistent Memory

Most persistent applications stem from traditional storage system workloads (databases and file systems), which require persistent memory [1, 2, 16, 17, 88, 90, 92] to support crash consistency [6], i.e., the **persistence property**. The persistence property guarantees that the critical data (e.g., database records, files, and the corresponding metadata) stored in nonvolatile devices retains a consistent state in case of power loss or a program crash, even when all the data in volatile devices may be lost. Achieving persistence in BA-NVM is nontrivial, due to the presence of volatile processor caches and memory write reordering performed by the write-back caches and memory controllers. For instance, a power outage may occur while a persistent application is inserting a node to a linked list stored in BA-NVM. Processor caches and memory controllers may reorder

[3]The recently developed BLISS scheduler [84] was shown to be more effective than TCM while providing low cost. Even though we do not evaluate BLISS, it also does not take into account the nature of interference caused by persistent applications.

the write requests, writing the pointer into BA-NVM before writing the values of the new node. The linked list can lose consistency with dangling pointers, if values of the new node remaining in processor caches are lost due to power outage, which may lead to unrecoverable data corruption. To avoid such inconsistency problems, most persistent memory designs borrow the ACID (atomicity, consistency, isolation, and durability) concepts from the database and file system communities [17, 54, 88, 90, 92]. Enforcing these concepts, as explained below, leads to additional memory requests, which affect the memory access behavior of persistent applications.

Versioning and Write Ordering. While durability can be guaranteed by BA-NVMs' non-volatile nature, atomicity and consistency are supported by storing multiple versions of the same piece of data and carefully controlling the order of writes into persistent memory (please refer to prior studies for details [17, 54, 88, 90, 92]). Figure 1 shows a persistent tree data structure as an example to illustrate the different methods to maintain versions and ordering. Assume nodes N_3 and N_4 are updated. We discuss two commonly-used methods to maintain multiple versions and ordering. The first one is *redo logging* [16, 90]. With this method, new values of the two nodes, along with their addresses, are written into a log ($\log N_3'$ and $\log N_4'$) before their original locations are updated in memory (Figure 1(a)). If a system loses power before logging is completed, persistent memory can always recover, using the intact original data in memory. A memory barrier is employed between the writes to the log and writes to the original locations in memory. This ordering control, with enough information kept in the log, ensures that the system can recover to a consistent state even if it crashes before all original locations are updated. The second method, illustrated in Figure 1(b), is the notion of *shadow updates* (copy-on-write) [17, 88]. Instead of storing logs, a temporary data buffer is allocated to store new values (shadow copies) of the nodes. Note that the parent node N_1 is also shadow-copied, with the new pointer N_1' pointing to the shadow copies N_3' and N_4'. Ordering control (shown as a memory barrier in Figure 1(b)) ensures that the root pointer is not updated until writes to the shadow copies are completed in persistent memory.

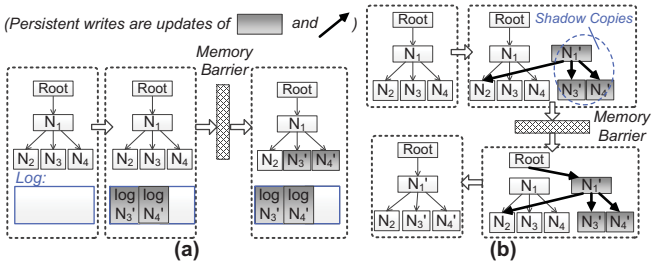

Fig. 1. Example persistent writes with (a) redo logging and (b) shadow updates, when nodes N3 and N4 in a tree data structure are updated.

Relaxed Persistence. Strict persistence [53, 54, 70] requires maintaining the program order of every write request, even within a single log update. Pelley *et al.* recently introduced a relaxed persistence model to minimize the ordering control to buffer and coalesce writes to the same data [70].[4] Our design adopts their relaxed persistence model. For example, we only enforce the ordering between the writes to shadow copies and to the root pointer, as shown in Figure 1(b). Another recent work, *Kiln* [92] relaxed versioning, eliminating the use of logging or shadow updates by implementing a nonvolatile last-level cache (NV cache). However, due to the limited capacity and associativity of the NV cache, the design cannot efficiently accommodate large-granularity persistent updates in database and file system applications. Consequently, we envision that logging, shadow updates, and Kiln-like designs will coexist in persistent memory designs in the near future.

[4]More recently, Lu et al. [54] proposed the notion of *loose-ordering consistency*, which relaxes the ordering of persistent memory writes even more by performing them speculatively.

2.3. Memory Requests of Persistent Applications

Persistent Writes. We define the writes to perform critical data updates that need to be persistent (including updates to original data locations, log updates, and shadow-copy updates), as **persistent writes**. Each critical data update may generate an arbitrary number of persistent writes depending on the granularity of the update. For example, in a key-value store, an update may be the addition of a new value of several bytes, several kilobytes, several megabytes, or larger. Note that persistent memory architectures either typically flush persistent writes (i.e., dirty blocks) out of processor caches at the point of memory barriers, or implement persistent writes as uncacheable (UC) writes [54, 88, 90, 92].

Non-persistent Writes. Non-critical data, such as stacks and data buffers, are not required to survive system failures. Typically, persistent memory does not need to perform versioning or ordering control over these writes. As such, persistent applications not only perform persistent writes but also non-persistent writes as well.

Reads. Persistent applications also perform reads of in-flight persistent writes and other independent reads. Persistent memory can relax the ordering of independent reads without violating the persistence requirement. However, doing so can impose substantial performance penalties (Section 3.2). Reads of in-flight persistent updates need to wait until these persistent writes arrive at BA-NVMs. Conventional memory controller designs provide read-after-write ordering by servicing reads of in-flight writes from write buffers. With volatile memory, such a behavior does not affect memory consistency. With nonvolatile memory, however, power outages or program crashes can destroy in-flight persistent writes before they are written to persistent memory. Speculative reads of in-flight persistent updates can lead to incorrect ordering and potential resulting inconsistency, because if a read has already gotten the value of an in-flight write that would disappear on a crash, wrong data may eventually propagate to persistent memory as a result of the read.

3. Motivation: Handling Persistent Memory Accesses

Conventional memory scheduling schemes are designed based on the assumption that main memory is used as working memory, i.e., a file cache for storage systems. This assumption no longer holds when main memory also supports data persistence, by accommodating persistent applications that access memory differently from traditional non-persistent applications. This is because persistent memory writes have different consistency requirements than working memory writes, as we described in Sections 2.2 and 2.3. In this section, we study the performance implications caused by this different memory access behavior of persistent applications (Section 3.1), discuss the problems of directly adopting existing memory scheduling methods to handle persistent memory accesses (Section 3.2), and describe why naïvely extending past memory schedulers does not solve the problem (Section 3.3).

3.1. Memory Access Characteristics of Persistent Applications

An application's memory access characteristics can be evaluated using four metrics: *a*) *memory intensity*, measured as the number of

last-level cache misses per thousand instructions (MPKI) [19, 42]; *b*) *write intensity*, measured as the portion of write misses (WR%) out of all cache misses; *c*) *bank-level parallelism (BLP)*, measured as the average number of banks with outstanding memory requests, when at least one other outstanding request exists [50, 65]; *d*) *row-buffer locality (RBL)*, measured as the average hit rate of the row buffer across all banks [64, 77].

To illustrate the different memory access characteristics of persistent and non-persistent applications, we studied the memory accesses of three representative micro-benchmarks, *streaming*, *random*, and *KVStore*. *Streaming* and *random* [42, 61] are both memory-intensive, non-persistent applications, performing streaming and random accesses to a large array, respectively. They serve as the two extreme cases with dramatically different BLP and RBL. The persistent application *KVStore* performs inserts and deletes to key-value pairs (25-byte keys and 2K-byte values) of an in-memory B+ tree data structure. The sizes of keys and values were specifically chosen so that *KVStore* had the same memory intensity as the other two micro-benchmarks. We build this benchmark by implementing a redo logging (i.e., writing new updates to a log while keeping the original data intact) interface on top of STX B+ Tree [12] to provide persistence support. Redo logging behaves very similarly to shadow updates (Section 2.2), which perform the updates in a shadow version of the data structure instead of logging them in a log space. Our experiments (not shown here) show that the performance implications of *KVStore* with shadow updates are similar to those of *KVStore* with redo logging, which we present here.

Table 1 lists the memory access characteristics of the three micro-benchmarks running separately. The persistent application *KVStore*, especially in its persistence phase when it performs persistent writes, has three major discrepant memory access characteristics in comparison to the two non-persistent applications.

Table 1. Memory access characteristics of three applications running individually. The last row shows the memory access characteristics of *KVStore* when it performs persistent writes.

	MPKI	**WR%**	**BLP**	**RBL**
Streaming	100/High	47%/Low	0.05/Low	96%/High
Random	100/High	46%/Low	6.3/High	0.4%/Low
KVStore	100/High	77%/High	0.05/Low	71%/High
Persistence Phase (KVStore)	*675/High*	*92%/High*	*0.01/Low*	*97%/High*

1. High write intensity. While the three applications have the same memory intensity, *KVStore* has much higher write intensity than the other two. This is because each insert or delete operation triggers a redo log update, which appends a log entry containing the addresses and the data of the modified key-value pair. The log updates generate extra write traffic in addition to the original location updates.

2. Higher memory intensity with persistent writes. The last row of Table 1 shows that while the *KVStore* application is in its persistence phase (i.e., when it is performing persistent writes and flushing these writes out of processor caches), it causes greatly higher memory traffic (MPKI is 675). During this phase, writes make up almost all (92%) the memory traffic.

3. Low BLP and high RBL with persistent writes. *KVStore*, especially while performing persistent writes, has low BLP and high RBL. *KVStore*'s log is implemented as a circular buffer, similar to those used in prior persistent memory designs [90], by allocating (as much as possible) one or more contiguous regions in the physical address space. As a result, the log updates lead to consecutive writes to contiguous locations in the same bank, i.e., an access pattern that can be characterized as *streaming writes*. This makes *KVStore*'s write behavior similar to that of *streaming*'s reads: low BLP and high RBL. However, the memory bus can only service either reads or writes (to any bank) at any given time because the bus can be driven in only one direction [49], which causes a fundamental difference (and conflict) between handling streaming reads and streaming writes.

We conclude that the persistent writes cause persistent applications to have widely different memory access characteristics than non-persistent applications. As we show next, the high write intensity and low bank-level parallelism of writes in persistent applications cause a fundamental challenge to existing memory scheduler designs for two reasons: 1) the high write intensity causes frequent switching of the memory bus between reads and writes, causing bus turnaround delays, 2) the low write BLP causes underutilization of memory bandwidth while writes are being serviced, which delays any reads in the memory request buffer. These two problems become exacerbated when persistent applications run together with non-persistent ones, a scenario where both reads and persistent writes are frequently present in the memory request buffer.

3.2. Inefficiency of Prior Memory Scheduling Schemes

As we mentioned above, the memory bus can service either reads *or* writes (to any bank) at any given time because the bus can be driven in only one direction [49]. Prior memory controllers (e.g., [26, 27, 41, 42, 49, 63, 64, 65, 76, 77, 95]) buffer writes in a write queue to allow read requests to aggressively utilize the memory bus. When the write queue is full or is filled to a predefined level, the memory scheduler switches to a *write drain mode* where it drains the write queue either fully or to a predetermined level [49, 78, 83], in order to prevent stalling the entire processor pipeline. During the *write drain mode*, the memory bus can service only writes. In addition, switching into and out of the *write drain mode* from the *read mode* induces additional penalty in the DRAM protocol (called read-to-write and write-to-read turnaround delays, t_{RTW} and t_{WTR}, approximately 7.5ns and 15ns, respectively [43]) during which no read or write commands can be scheduled on the bus, causing valuable memory bus cycles to be wasted. Therefore, frequent switches into the *write drain mode* and long time spent in the *write drain mode* can significantly slow down reads and can harm the performance of read-intensive applications and the entire system [49].

This design of conventional memory schedulers is based on two assumptions, which are generally sound for non-persistent applications. First, reads are on the critical path of application execution whereas writes are usually not. This is sound when most non-persistent applications abound with read-dependent arithmetic, logic, and control flow operations and writes can be serviced from write buffers in caches and in the memory controller. Therefore, most prior memory scheduling schemes prioritize reads over writes. Second, applications are usually read-intensive, and memory controllers can delay writes without frequently filling up the write queues. Therefore, optimizing the performance of writes is not as critical to performance in many workloads as the write queues are large enough for such read-intensive applications.

Unfortunately, these assumptions no longer hold when persistent writes need to go through the same shared memory interface as non-persistent requests. First, the ordering control of persistent writes requires the serialization of the persistent write traffic to main memory (e.g., via the use of memory barriers, as described in Section 2.2). This causes the persistent writes, reads of in-flight persistent writes, and computations dependent on these writes (and

potentially all computations after the persistent writes, depending on the implementation) to be serialized. As such, *persistent writes are also on the critical execution path.* As a result, simply prioritizing read requests over persistent write requests can hurt system performance. Second, persistent applications are *write-intensive* as opposed to read-intensive. This is due to not only the persistent nature of data manipulation, which might lead to more frequent memory updates, but also the way persistence is guaranteed using multiple persistent updates (i.e., to the original location as well as the alternate version of the data in a redo log or a shadow copy, as explained in Section 2.2).

Because of these characteristics of persistent applications, existing memory controllers are inefficient in handling them concurrently with non-persistent applications. Figure 2 illustrates this inefficiency in a system that concurrently runs *KVStore* with either the *streaming* or the *random* application. This figure shows the fraction of memory access cycles that are spent due to delays related to bus turnaround between reads and writes as a function of the number of write queue entries.[5] The figure shows that up to 17% of memory bus cycles are wasted due to frequent bus turnarounds, with a commonly-used 64-entry write queue. We found that this is mainly because persistent writes frequently overflow the write queue and force the memory controller to drain the writes. Typical schedulers in modern processors have only 32 to 64 write queue entries to buffer memory requests [30]. Simply increasing the number of write queue entries in the scheduler is not a scalable solution [7].

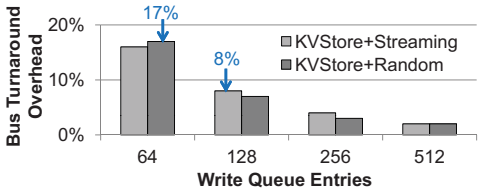

Fig. 2. Fraction of memory access cycles wasted due to delays related to bus turnaround between reads and writes.

In summary, conventional memory scheduling schemes, which prioritize reads over persistent writes, become inefficient when persistent and non-persistent applications share the memory interface. This causes relatively low performance and fairness (as we show next).

3.3. Analysis of Prior and Naïve Scheduling Policies

We have observed, in Section 2.3, that the persistent applications' (e.g., *KVStore*'s) writes behave similarly to streaming reads. As such, a natural idea would be to assign these persistent writes *the same priority* as read requests, instead of deprioritizing them below read requests, to ensure that persistent applications are not unfairly penalized. This is a naïve (yet simple) method of extending past schedulers to potentially deal with persistent writes.

In this section, we provide a case study analysis of fairness and performance of both prior schedulers (FR-FCFS [76, 77, 95] and TCM [42]) and naïve extensions of these prior schedulers (FRFCFS-modified and TCM-modified) that give equal priority to reads and persistent writes.[6] Figure 3 illustrates fairness and system performance of these schedulers for two workloads where *KVStore* is run together with *streaming* or *random*. To evaluate fairness, we consider both the individual slowdown of each application [48] and the maximum slowdown [20, 41, 42, 87] across both applications in a workload. We make several major observations.

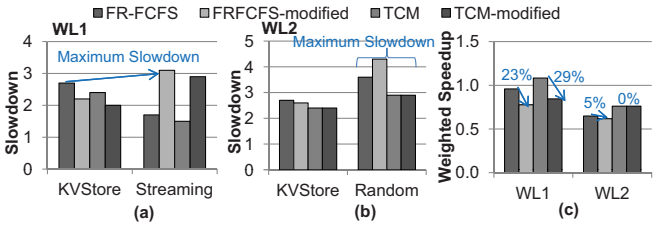

Fig. 3. Performance and fairness of prior and naïve scheduling methods.

Case Study 1 (WL1 in Figure 3(a) and (c)): When *KVStore* is run together with *streaming*, prior scheduling policies (FR-FCFS and TCM) unfairly slow down the persistent *KVStore*. Because these policies delay writes behind reads, and *streaming*'s reads with high row-buffer locality capture a memory bank for a long time, *KVStore*'s writes need to wait for long time periods even though they also have high row buffer locality. When the naïve policies are employed, the effect is reversed: FRFCFS-modified and TCM-modified reduce the slowdown of *KVStore* but increase the slowdown of *streaming* compared to FRFCFS and TCM. *KVStore* performance improves because, as persistent writes are the same priority as reads, its frequent writes are not delayed too long behind *streaming*'s reads. *Streaming* slows down greatly due to two major reasons. First, its read requests are interfered much more frequently with the write requests of *KVStore*. Second, due to equal read and persistent write priorities, the memory bus has to be frequently switched between persistent writes and streaming reads, leading to high bus turnaround latencies where no request gets scheduled on the bus. These delays slow down both applications but affect *streaming* a lot more because almost all accesses of *streaming* are reads and are on the critical path, and are affected by both read-to-write and write-to-read turnaround delays whereas *KVStore*'s writes are less affected by write-to-read turnaround delays. Figure 3(c) shows that the naïve policies greatly degrade overall system performance on this workload, even though they improve *KVStore*'s performance. We find this system performance degradation is mainly due to the frequent bus turnarounds.

Case Study 2 (WL2 in Figure 3(b) and (c)): *KVStore* and *random* are two applications with almost exactly opposite BLP, RBL, and write intensity. When these two run together, *random* slows down the most with all of the four evaluated scheduling policies. This is because *random* is more vulnerable to interference than the mostly-streaming *KVStore* due to its high BLP, as also observed in previous studies [42]. FRFCFS-modified slightly improves *KVStore*'s performance while largely degrading *random*'s performance due to the same reason described for WL1. TCM-modified does not significantly affect either application's performance because three competing effects end up canceling any benefits. First, TCM-modified ends up prioritizing the random-access *random* over streaming *KVStore* in some time intervals, as it is aware of the high vulnerability of *random* due to its high BLP and low RBL. Second, at other times, it prioritizes the frequent persistent write requests of *KVStore* over read requests of *random* due to the equal priority of reads and persistent writes. Third, frequent bus turnarounds (as discussed above for WL1) degrade both applications' performance. Figure 3(c) shows that the naïve policies slightly degrade or not affect overall system performance on this workload.

[5] Section 6 explains our system setup and methodology.

[6] Note that we preserve all the other ordering rules of FR-FCFS and TCM in FRCFCS-modified and TCM-modified. Within each prioritization level, reads and persistent writes are prioritized over non-persistent writes. For example, with FRCFCS-modified, the highest priority requests are row-buffer-hit read and persistent write requests, second highest priority requests are row-buffer-hit non-persistent write requests.

3.4. Summary and Our Goal

In summary, neither conventional scheduling policies nor their naïve extensions that take into account persistent writes provide high fairness and high system performance. This is because they lead to 1) frequent entries into *write drain mode* due to high intensity of persistent writes, 2) resulting frequent bus turnarounds between read and write requests that cause wasted bus cycles, and 3) memory bandwidth underutilization during *write drain mode* due to low BLP of persistent writes. These three problems are pictorially illustrated in Figure 4(a) and (b), which depict the service timeline of memory requests with conventional scheduling and its naïve extension. This illustration shows that 1) persistent writes heavily access Bank-1, leading to high bandwidth underutilization with both schedulers, 2) both schedulers lead to frequent switching between reads and writes, and 3) the naïve scheduler delays read requests significantly because it prioritizes persistent writes, and it does not reduce the bus turnarounds. Our evaluation of 18 workload combinations (in Section 7) shows that various conventional and naïve scheduling schemes lead to low system performance and fairness, due to these three reasons. Therefore, a new memory scheduler design is needed to overcome these challenges and provide high performance and fairness in a system where the memory interface is shared between persistent and non-persistent applications. Our goal in this work is to design such a scheduler (Section 4).

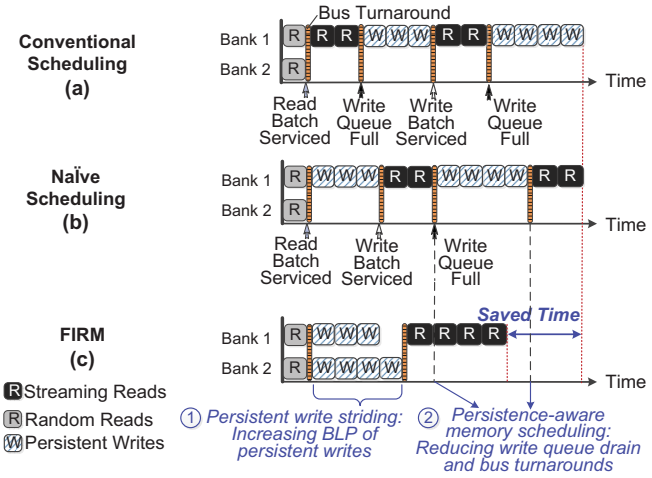

Fig. 4. Example comparing conventional, naïve, and proposed schemes.

4. FIRM Design

Overview. We propose FIRM, a memory control scheme that aims to serve requests from persistent and non-persistent applications in a fair and high throughput manner at the shared memory interface. FIRM introduces two novel design principles to achieve this goal, which are illustrated conceptually in Figure 4(c). First, *persistent write striding* ensures that persistent writes to memory have high BLP such that memory bandwidth is well-utilized during the *write drain mode*. It does so by ensuring that consecutively-issued groups of writes to the log or shadow copies in persistent memory are mapped to different memory banks. This reduces not only the duration of the *write drain mode* but also the frequency of entry into *write drain mode* compared to prior methods, as shown in Figure 4(c). Second, *persistence-aware memory scheduling* minimizes the frequency of write queue drains and bus turnarounds by scheduling the queued up reads and writes in a fair manner. It does so by balancing the amount of time spent in *write drain mode* and *read mode*, while ensuring that the time spent in each mode is long enough such that the wasted cycles due to bus turnaround delays are minimized. *Persistence-aware memory scheduling* therefore reduces: 1) the latency of servicing the persistent writes, 2) the amount of time persistent writes block outstanding reads, and 3) the frequency of entry into *write queue drain* mode. The realization of these two principles leads to higher performance and efficiency than conventional and naïve scheduler designs, as shown in Figure 4.

FIRM design consists of four components: 1) *request batching*, which forms separate batches of read and write requests that go to the same row, to maximize row buffer locality, 2) *source categorization*, which categorizes the request sources for effective scheduling by distinguishing various access patterns of applications, 3) *persistent write striding*, which maximizes BLP of persistent requests, and 4) *persistence-aware memory scheduling*, which maximizes performance and fairness by appropriately adjusting the number of read and write batches to be serviced at a time. Figure 5(a) depicts an overview of the components, which we describe next.

4.1. Request Batching

The goal of request batching is to group together the set of requests to the same memory row from each source (i.e., process or hardware thread context, as described below in Section 4.2). Batches are formed per source, similarly to previous work [7, 65], separately for reads and writes. If scheduled consecutively, all requests in a read or write batch (except for the first one) will hit in the row buffer, minimizing latency and maximizing memory data throughput. A batch is considered to be formed when the next memory request in the request buffer of a source is to a different row [7].

4.2. Source Categorization

To apply appropriate memory control over requests with various characteristics, FIRM dynamically classifies the sources of memory requests into four: *non-intensive*, *streaming*, *random*, *persistent*. A source is defined as a process or thread during a particular time period, when it is generating memory requests in a specific manner. For example, a persistent application is considered a *persistent* source when it is performing persistent writes. It may also be a *non-intensive*, a *streaming*, or a *random* source in other time periods.

FIRM categorizes sources on an interval basis. At the end of an interval, each source is categorized based on its memory intensity, RBL, BLP, and persistence characteristics during the interval, predicting that it will exhibit similar behavior in the next interval.[7]

The main new feature of FIRM's source categorization is its detection of a *persistent* source (inspired by the discrepant characteristics of persistent applications described in Section 2.3). Table 2 depicts the rules FIRM employs to categorize a source as persistent. FIRM uses program hints (with the software interface described in Section 5) to determine whether a hardware context belongs to a persistent application. This ensures that a non-persistent application does not get classified as a persistent source. If a hardware context belonging to such an application is generating write batches that are larger than a pre-defined threshold (i.e., has an average write batch size greater than 30 in the previous interval) and if it inserts memory barriers between memory requests (i.e., has inserted at least one memory barrier between write requests in the previous interval), FIRM categorizes it as a *persistent* source.

[7] We use an interval size of one million cycles, which we empirically find to provide a good tradeoff between prediction accuracy, adaptivity to workload behavior, and overhead of categorization.

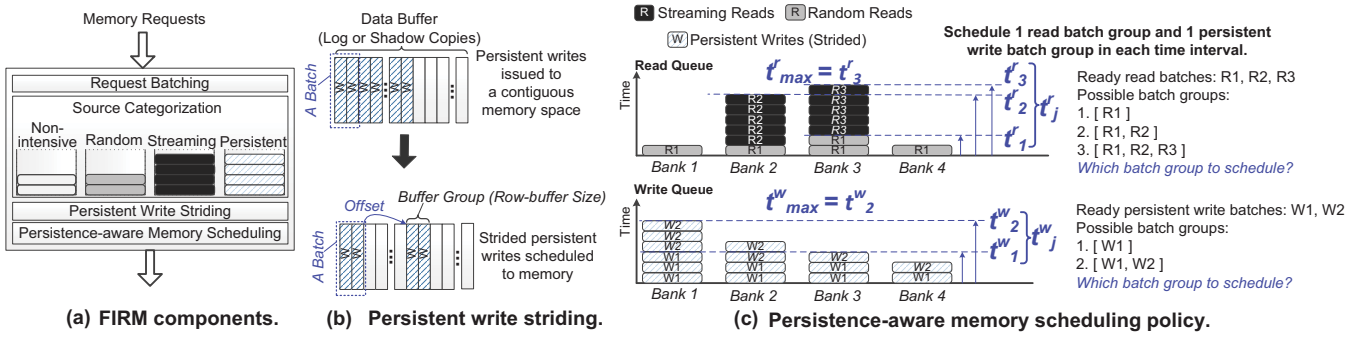

Fig. 5. Overview of the FIRM design and its two key techniques.

Table 2. Rules used to identify persistent sources.

A thread is identified as a persistent source, if it	
1:	belongs to a persistent application;
2:	is generating write batches that are larger than a pre-defined threshold in the past interval;
3:	inserts memory barriers between memory requests.

Sources that are not persistent are classified into *non-intensive*, *streaming*, and *random* based on three metrics: MPKI (memory intensity), BLP, RBL. This categorization is inspired by previous studies [42, 63], showing varying characteristics of such sources. A *non-intensive* source has low memory intensity. We identify these sources to prioritize their batches over batches of other sources; this maximizes system performance as such sources are latency sensitive [42]. *Streaming* and *random* sources are typically read intensive, having opposite BLP and RBL characteristics (Table 1).[8] This *streaming* and *random* source classification is used later by the underlying scheduling policy FIRM borrows from past works to maximize system performance and fairness (e.g., TCM [42]).

4.3. Persistent Write Striding

The goal of this mechanism is to reduce the latency of servicing consecutively-issued persistent writes by ensuring they have high BLP and thus fully utilize memory bandwidth. We achieve this goal by *striding the persistent writes across multiple memory banks* via hardware or software support.

The basic idea of *persistent write striding* is simple: instead of mapping consecutive groups of row-buffer-sized persistent writes to consecutive row-buffer-sized locations in a persistent data buffer (that is used for the redo log or shadow copies in a persistent application), which causes them to map to the same memory bank, change the mapping such that they are strided by an offset that ensures they map to different memory banks.

Figure 5(b) illustrates this idea. A persistent application can still allocate a contiguous memory space for the persistent data buffer. Our method maps the accesses to the data buffer to different banks in a strided manner. Contiguous persistent writes of less than or equal to the row-buffer size are still mapped to contiguous data buffer space with of a row buffer size (called a "buffer group") to achieve high RBL. However, contiguous persistent writes beyond the size of the row-buffer are strided by an *offset*. The value of the offset is determined by the position of bank index bits used in the physical address mapping scheme employed by the memory controller. For example, with the address mapping scheme in Figure 6, the offset should be 128K bytes if we want to fully utilize all eight banks with persistent writes (because a contiguous memory chunk of 16KB gets mapped to the same bank with this address mapping scheme, i.e., the memory interleaving granularity is 16KB across banks). This persistent write striding mechanism can be implemented in either the memory controller hardware or a user-mode library, as we describe in Section 5.

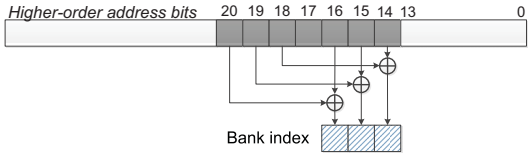

Fig. 6. Physical address to bank mapping example.

Note that the *persistent write striding* mechanism provides a deterministic (re)mapping of persistent data buffer physical addresses to physical memory addresses in a strided manner. The remapped physical addresses will not exceed the boundary of the original data buffer. As a result, re-accessing or recovering data at any time from the persistent data buffer is not an issue: all accesses to the buffer go through this remapping.

Alternative Methods. Note that commodity memory controllers randomize higher-order address bits to minimize bank conflicts (Figure 6). However, they can still fail to map persistent writes to different banks because as we showed in Section 3.1, persistent writes are usually streaming and hence they are likely to map to the same bank. It is impractical to improve the BLP of persistent writes by aggressively buffering them due to two reasons: 1) The large buffering capacity required. For example, we might need a write queue as large as 128KB to utilize all eight banks of a DDR3 channel with the address mapping shown in Figure 6. 2) The region of concurrent contiguous writes may not be large enough to cover multiple banks (i.e., there may not be enough writes present to different banks). Alternatively, kernel-level memory access randomization [69] may distribute writes to multiple banks during persistent application execution. However, the address mapping information can be lost when the system reboots, leaving the BA-NVM with unrecoverable data. Finally, it is also prohibitively complex to randomize the bank mapping of *only* persistent writes by choosing a different set of address bits as their bank indexes, i.e., maintaining multiple address mapping schemes in a single memory system. Doing so requires complex bookkeeping mechanisms to ensure correct mapping of memory addresses. For these very reasons, we have developed the *persistent write striding* technique we have described.

[8] In our experiments, a hardware context is classified as *non-intensive* if its MPKI< 1. A hardware context is classified as *streaming* if its MPKI>1, BLP<4 and RBL>70%. All other hardware contexts that are not persistent are classified as *random*.

4.4. Persistence-Aware Memory Scheduling

The goal of this component is to minimize write queue drains and bus turnarounds by intelligently partitioning memory service between reads and persistent writes while maximizing system performance and fairness. To achieve this multi-objective goal, FIRM operates at the batch granularity and forms a schedule of read and write batches of different source types: *non-intensive*, *streaming*, *random*, and *persistent*. To maximize system performance, FIRM prioritizes *non-intensive read batches* over all other batches.

For the remaining batches of requests, FIRM employs a new policy that determines 1) how to group read batches and write batches and 2) when to switch between servicing read batches and write batches. FIRM does this in a manner that balances the amount of time spent in *write drain mode* (servicing write batches) and *read mode* (servicing read batches) in a way that is proportional to the read and write demands, while ensuring that time spent in each mode is long enough such that the wasted cycles due to bus turnaround delays are minimized. When the memory scheduler is servicing read or persistent write batches, in *read mode* or *write drain mode*, the scheduling policy employed can be any of the previously-proposed memory request scheduling policies (e.g., [26, 41, 42, 49, 64, 65, 76, 77, 95]) and the ordering of *persistent write batches* is fixed by the ordering control of persistent applications. The key novelty of our proposal is *not* the particular prioritization policy between requests, but the mechanism that determines how to group batches and when to switch between *write drain mode* and *read mode*, which we describe in detail next.[9]

To minimize write queue drains, FIRM schedules reads and persistent writes within an interval in a round-robin manner with the memory bandwidth (i.e., the time interval) partitioned between them based on their demands. To prevent frequent bus turnarounds, FIRM schedules a group of batches in one bus transfer direction before scheduling another group of batches in the other direction. Figure 5(c) illustrates an example of this persistence-aware memory scheduling policy. Assume, without loss of generality, that we have the following batches ready to be scheduled at the beginning of a time interval: a random read batch *R1*, two streaming read batches *R2* and *R3*, and two (already-strided) persistent write batches *W1* and *W2*. We define a *batch group* as a group of batches that will be scheduled together. As illustrated in Figure 5(c), the memory controller has various options to compose the read and write batch groups. This figure shows three possible batch groups for reads and two possible batch groups for writes. These possibilities assume that the underlying memory request scheduling policy dictates the order of batches within a batch group. Our proposed technique thus boils down to determining how many read or write batches to be grouped together to be scheduled in the next *read mode* or *write drain mode*.

We design a new technique that aims to satisfy the following two goals: 1) servicing the two batch groups (read and write) consumes durations proportional to their demand, 2) the total time spent servicing the two batch groups is much longer than the bus turnaround time. The first goal is to prevent the starvation of either reads or persistent writes, by fairly partitioning the memory bandwidth between them. The second goal is to maximize performance by ensuring minimal amount of time is wasted on bus turnarounds.

Mathematically, we formulate these two goals as the following two inequalities:

$$\begin{cases} \frac{t^r}{t^w} \approx \frac{t^r_{max}}{t^w_{max}} \\ \frac{t_{RTW}+t_{WTR}}{t^r+t^w} \leq \mu_{turnaround} \end{cases} \quad (1)$$

where t^r and t^w are the times to service a read and a persistent write batch group, respectively (Figure 5(c)). They are the maximum service time for the batch group at any bank i:

$$\begin{cases} t^r = \max_i\{H^r_i t^{rhit}_i + M^r_i t^{rmiss}_i\}, \\ t^w = \max_i\{H^w_i t^{whit}_i + M^w_i t^{wmiss}_i\} \end{cases} \quad (2)$$

where t^{rhit}, t^{whit}, t^{rmiss}, and t^{wmiss} are the times to service a row buffer hit/miss read/write request; H^r_i and H^w_i are the number of row-buffer read/write hits; M^r_i and M^w_i are row-buffer read/write misses. t^r_{max} and t^w_{max} are the maximum times to service all the in-flight read and write write requests (illustrated in Figure 5 (c)). $\mu_{turnaround}$ is a user-defined parameter to represent the maximum tolerable fraction of bus turnaround time out of the total service time of memory requests.

The goal of our mechanism is to group read and write batches (i.e., form read and write batch groups) to be scheduled in the next *read mode* and *write drain mode* in a manner that satisfies Equation 1. Thus, the technique boils down to selecting from the set of possible read/write batch groups such that they satisfy t^r_{next} (the duration of the next *read mode*) and t^w_{next} (the duration of the next *write drain mode*) as indicated by the constraints in Equation 3 (which is obtained by solving the inequality in Equation 1). Our technique, Algorithm 1, forms a batch group that has a minimum service duration that satisfies the constraint on the right hand side of Equation 3.[10]

$$\begin{cases} t^r_{next} = \min_j t^r_j, \quad t^r_j \geq \frac{(t_{RTW}+t_{WTR})/\mu_{turnaround}}{1+t^w_{max}/t^r_{max}} \\ t^w_{next} = \min_j t^w_j, \quad t^w_j \geq \frac{(t_{RTW}+t_{WTR})/\mu_{turnaround}}{1+t^r_{max}/t^w_{max}} \end{cases} \quad (3)$$

Algorithm 1 Formation of read and write batch groups.

Input: t_{RTW}, t_{WTR}, and $\mu_{turnaround}$.
Output:
The read batch group to be scheduled, indicated by t^r_{next};
The persistent write batch group to be scheduled, indicated by t^w_{next}.
Initialization:
$k^r \leftarrow$ number of read batch groups;
$k^w \leftarrow$ number of persistent write batch groups;
for $j \leftarrow 0$ to $k^r - 1$ **do**
  Calculate t^r_j with Equation 2;
end for
for $j \leftarrow 0$ to $k^w - 1$ **do**
  Calculate t^w_j with Equation 2;
end for
$t^r_{max} \leftarrow \max_{j=0}^{k^r-1} t^r_j$; $t^w_{max} \leftarrow \max_{j=0}^{k^w-1} t^w_j$;
Calculate t^r_{next} and t^w_{next} with Equation 3;

5. Implementation

5.1. Software-Hardware Interface

FIRM software interface provides memory controllers with the required information to identify persistent sources during the source categorization stage. This includes 1) the identification of the persistent application, 2) the communication of the execution of memory

[9] Note that most previous memory scheduling schemes focus on read requests and do not discuss how to handle switching between read and write modes in the memory controller, implicitly assuming that reads are prioritized over writes until the write queue becomes full or close to full [49].

[10] This algorithm can be invoked only once at the beginning of each interval to determine the (relative) durations of consecutive *read* and *write drain* modes for the interval.

barriers. We offer programmers two options to define persistent applications: 1) declaring an entire application as persistent, or 2) specifying a thread of an application to perform persistent updates, i.e., a *persistent thread*. With the first option, programmers can employ the following software interface (similar to Kiln [92]):

```
#pragma persistent_memory
```

With the second option, programmers can annotate a particular thread with a `persistent` attribute when they create the thread. The software interface is translated to ISA instructions with simple modifications to compilers and the ISA. Similar ISA extensions have been employed by previous studies [32, 35, 92]. Once a processor reads software's input of designating a persistent application/thread, it signals each memory controller by setting *persistent thread registers* to indicate the presence of such an application/thread. Each persistent thread register stores $log_2 N_{hwthreads}$-bit hardware thread identifiers (IDs).

The mechanism to detect memory barriers depends on the CPU architecture and is already present in many processors [56].[11] The processor communicates the execution of a barrier needs to the memory controller.

5.2. Implementing Persistent Write Striding

Our persistent write striding mechanism can be implemented by modifying memory controller hardware or the logging/shadow update operations in a user-mode library (e.g., employed by Mnemosyne [90]). The two methods trade off between hardware overhead and low-level software implementation effort.

To implement persistent write striding in the memory controller, we employ a pair of registers and a counter for each persistent hardware thread. The two registers, referred to as *start_address* and *end_address*, initially record the starting and the end addresses of a contiguous persistent memory region (a data buffer storing the redo log or shadow copies) allocated by a persistent application/thread and currently being accessed. With the two registers, the memory controller can identify the boundary of the data buffer when the persistent application starts writing to it. These registers are reset when a new data buffer is allocated. A 6-bit counter, referred to as the *intra-group index*, is used to determine when a buffer group (Figure 5(b)) is fully occupied. It records the number of appended log or shadow updates within a group. When the value of the intra-group index reaches the end of a buffer group, the memory controller will map the coming write requests to the next group in the data buffer by striding the physical address with an offset. At this time point, we also update the start_address register with the starting address of its neighboring buffer group. When the next strided physical address exceeds the end of the entire data buffer, the corresponding persistent update will start to write to the first empty buffer group indicated by the start_address register.

We can avoid the above hardware overhead by implementing persistent write striding in a user-mode library. For example, with logging-based persistent applications, we can modify the `log_append()` function(s) [90] to stride for an offset with each log append request defined by programmers.

5.3. Tracking Memory Access Characteristics

Table 3 lists the parameters to be tracked for each executing thread at the beginning of each time interval. We employ a set of counters and logic in memory controllers to collect the parameters. Some of the counters have already been implemented in previous work [42, 65], which also need to track MPKI, BLP, and RBL. Our design adds a set of request counters in each memory controller, each belonging to a hardware thread context, to track the read and write batch sizes of each source. We reset the counters at the beginning of each time interval, after the memory request scheduling decision is made.

Table 3. Parameters tracked for each executing thread.

Parameters	Usage
MPKI, BLP, RBL	To categorize non-intensive, streaming, and random sources
Size of each write batch	To categorize persistent sources; to calculate t^w (Section 4.4)
Size of each read batch	To calculate t^r (Section 4.4)

5.4. Area Overhead

Each memory controller maintains its own per-thread hardware counters and registers. Thus, it can independently make scheduling decisions for its local requests. Consequently, our design does not require a centralized arbiter to coordinate all the controllers. Table 4 lists the storage required by registers and counters described in this section, in each memory controller. FIRM requires adding 1192 (2400) bits in each memory controller, if the processor has 8 (16) hardware threads. The complexity of the scheduling algorithm is comparable to those of prior designs [42, 65].

Table 4. Storage required by FIRM in each memory controller. $N_{hwthreads}$ is the number of hardware contexts.

Register/Counter Name	Storage (bits)
Persistent thread registers	$log_2 N_{hwthreads} \times N_{hwthreads}$
Start_address registers	$64 \times N_{hwthreads}$
End_address registers	$64 \times N_{hwthreads}$
Intra-group index counters	$6 \times N_{hwthreads}$
Request counters	$2 \times 6 \times N_{hwthreads}$

6. Experimental Setup

6.1. Simulation Framework

We conducted our experiments using McSimA+ [4], a Pin-based [55] cycle-level multi-core simulator. Table 5 lists the parameters of the processor and memory system used in our experiments. Each processor core is similar to one of the Intel Core i7 cores [3]. The processor incorporates SRAM-based volatile private and shared caches. The cores and L3 cache banks communicate with each other through a crossbar interconnect. The processor employs a two-level hierarchical directory-based MESI protocol to maintain cache coherence. The BA-NVM is modeled as off-chip DIMMs compatible with DDR3.[12] Except for sensitivity studies, we conservatively employ the worst-case timing parameters of the BA-NVM generated by NVSim [24].

6.2. Benchmarks

Table 6 lists the characterization results of our benchmarks when run alone. We select seven non-persistent applications with different memory intensity, BLP, and RBL: four single-threaded applications from SPEC CPU2006 [82] (*mcf*, *lbm*, *leslie3d*, and *povray*) and three multithreaded benchmarks from PARSEC 3.0 [11] (*x264*, *ferret*, and

[11] For example, x86 processors use the memory fence instruction `mfence`, which prevents progress of the thread until the store buffer is drained [56].

[12] Everspin recently launched DDR3-compatible STT-MRAM components [36], which transfer data at a speed comparable to current DDR3-1600.

Table 5. Parameters of the evaluated multi-core system.

Processor	Similar to Intel Core i7 / 22 nm
Cores	4 cores, 2.5GHz, 2 threads per core
L1 Cache (Private)	64KB, 4-way, 64B lines, 1.6ns latency
L2 Cache (Private)	256KB, 8-way, 64B lines, 4.4ns latency
L3 Cache (Shared)	Multi-banked, 2MB/core, 16-way, 64B lines, 10ns latency
Memory Controller	64-/64-entry read/write queues
BA-NVM DIMM	STT-MRAM, 8GB, 8 banks, 2KB row, 36ns row-buffer hit, 65/76ns read/write row-buffer conflict

dedup). Currently, persistent applications are not available in public benchmark suites. Similar to recent studies on persistent memory [16, 38, 52, 70, 92], we constructed three benchmarks with persistent applications on common data structures used in file systems and databases. Each persistent application searches for a node with a randomly generated key in three different data structures (a B+ tree, a hash table, and an array), deletes the node if it exists, and inserts it otherwise. Redo logging is used to maintain versioning, although FIRM is applicable to shadow copies, too, since its techniques are orthogonal to versioning and write order control mechanisms.

Table 6. Benchmarks. The second numbers in the last three rows belong to the persistence phase.

	MPKI	**WR%**	**BLP**	**RBL**
mcf	32.0	25.6%	6.0	41.1%
lbm	28.2	42.0%	2.8	78.7%
leslie3d	15.7	4.0%	1.7	90.8%
povray	0.1	6.0%	1.2	77.6%
x264	22.5	26.2%	0.7	87%
ferret	12.6	13.9%	4.6	58.2%
dedup	20.2	20.8%	1.6	90.1%
Btree	26.2, 71.9	22.0%, 92.2%	0.4, 0.2	57.6%, 97.6%
Hash	37.1, 62.7	42.6%, 95.3%	0.2, 0.1	95.6%, 98.0%
SPS	11.0, 60.5	35.2%, 89.0%	0.2, 0.1	65.1%, 93.4%

Table 7 shows the evaluated workload combinations. *W1* trough *W6* are non-persistent workloads with three threads. *W1* through *W3* consist of single-threaded workloads with different fractions of streaming and random memory accesses. *W4* through *W6* each runs four threads. We constructed 18 workloads mixed by persistent and non-persistent applications to be used in our experiments: *W1* through *W3* are combined with persistent applications running one thread (each application is mapped to one processor core). *W4* through *W6* are combined with persistent applications running four threads, leading to a total of eight threads mapped to four two-threaded cores.

6.3. Metrics

We evaluate system throughput with weighted speedup [29, 80]:

$$WeightedSpeedup = \sum_i \frac{Throughput_i^{shared}}{Throughput_i^{alone}}$$

For SPEC CPU2006, $Throughput_i$ is calculated as instruction throughput, i.e., number of executed instructions per cycle. The throughput of *x264* is calculated by frame rate. The throughput of the rest of PARSEC benchmarks and persistent applications is calculated by operation throughput, e.g., the number of completed search, insert, and delete operations per cycle. We evaluate unfairness using maximum slowdown [20, 41, 42, 87]:

$$MaximumSlowdown = \max_i \frac{Throughput_i^{alone}}{Throughput_i^{shared}}$$

7. Results

We present our evaluation results and their analyses. We set $\mu_{turnaround} = 2\%$ in all our experiments.

7.1. Performance and Fairness

We demonstrate system throughput and fairness of FIRM by comparing it with various prior memory scheduling mechanisms in Figure 7. *FR-FCFS* [76, 77, 95], *PAR-BS* [65] and *TCM* [42] are conventional memory scheduling schemes. *TCM-modified* is a naïve extended scheduling policy discussed in Section 3.3. *NVMDuet* is a memory scheduling policy recently proposed for persistent memory systems [52]. It resolves the contention between *only* persistent and non-persistent writes. To illustrate the performance and fairness benefits of FIRM's two components, *persistent write striding* and *persistence-aware memory scheduling*, we collect the results of applying only persistent write striding (*FIRM-Strided*) and both (*FIRM-Strided-Scheduling*) separately. In both cases, we assume the request prioritization policy in FIRM is TCM, which offers the best performance and fairness across various conventional and naïve schemes. We make three major observations.

First, because writes are already deprioritized in our baseline schedulers, *NVMDuet*, which mainly tackles the contention between persistent and non-persistent writes, can improve system performance by only 1.1% (a maximum of 2.6%) and reduces unfairness by 1.7% on average compared to *TCM* (see Section 9 for more detail). It shifts the most slowed down application from non-persistent ones to the persistent ones in workload combinations *Btree-W1*, *Btree-W2*, *SPS-W1*, and *SPS-W2*.[13] Second, FIRM-Strided can effectively accelerate persistent writes by scheduling them to multiple banks. It increases system performance by 10.1% on average compared to *TCM*.Third, FIRM, with both of its components, provides the highest performance and fairness of *all* evaluated scheduling policies on *all* workloads. Hence, the benefits of FIRM are consistent. Overall, compared to *TCM*, FIRM increases average system performance and fairness by 17.9% and 23.1%, respectively. We conclude that FIRM is the most effective policy in mitigating the memory contention between persistent and non-persistent applications.

7.2. Bus Turnaround Overhead

Figure 8 compares the bus turnaround overhead (i.e., the fraction of total memory access time spent on bus turnarounds) of FIRM and TCM. With conventional and naïve scheduling schemes, including TCM, the frequency of bus turnarounds is determined by the read/write request batch sizes and the frequency of write queue drains. Both smaller request batches and frequent write queue drains lead to frequent bus turnarounds. We observe that FIRM's two components can effectively reduce the bus turnaround overhead. FIRM-Strided reduces the bus turnaround overhead of a subset of workloads, where the frequency of bus turnarounds is dictated by the frequency of write queue drain. On average, FIRM-Strided reduces bus turnaround overhead by 12% over *TCM*. Our persistence-aware memory scheduling policy can reduce the bus turnaround frequency even further by dividing bandwidth carefully between read and write batch groups. Therefore, FIRM-Strided-Scheduling, which combines the two FIRM components, reduces bus turnaround overhead by 84% over *TCM*. We conclude that FIRM can effectively minimize wasted cycles due to bus turnarounds caused by ineffective handling of persistent applications by past memory schedulers.

[13] The low performance and fairness benefits are consistent, so we do not show results of *NVMDuet* in the rest of this section.

Table 7. Workloads mixed with various persistent and non-persistent applications.

Non-persistent Workloads				Mix of Persistent and Non-persistent Applications						
Index	Benchmarks	Index	Benchmark		W1	W2	W3	W4	W5	W6
W1	mcf, lbm, povray	W4	x264	**Btree**	Btree-W1	Btree-W2	Btree-W3	Btree-W4	Btree-W5	Btree-W6
W2	mcf, lbm, leslie3d	W5	ferret	**Hash**	Hash-W1	Hash-W2	Hash-W3	Hash-W4	Hash-W5	Hash-W6
W3	mcf(2), leslie3d	W6	dedup	**SPS**	SPS-W1	SPS-W2	SPS-W3	SPS-W4	SPS-W5	SPS-W6

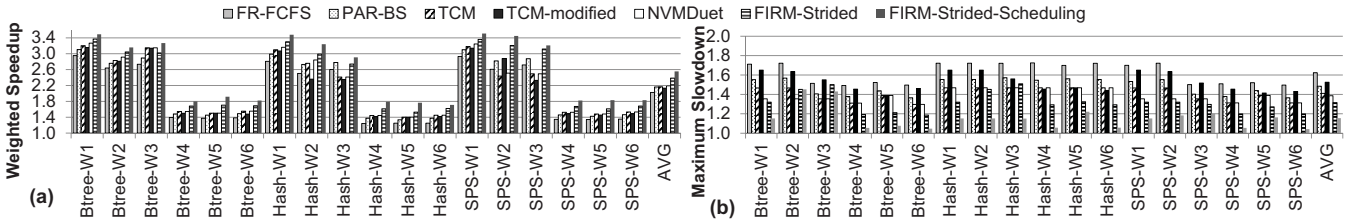

Fig. 7. System performance and fairness of FIRM versus various memory scheduling schemes.

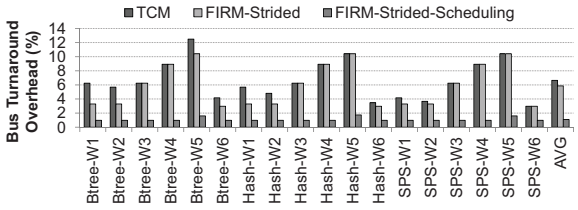

Fig. 8. Bus turnaround overhead.

7.3. Sensitivity to BA-NVM Latency

Recently published results on BA-NVM latencies [15, 39, 68, 86] suggest that the range of read latency is between $32ns$ and $120ns$ across various BA-NVM technologies, and the range of write latency is between $40ns$ and $150ns$. Therefore, we sweep the read and write row-buffer conflict latencies from $0.5\times$ to $2\times$ of the parameters listed in Table 5. For example, *R2W2* represents doubled read and write latencies (for row-buffer conflicts). As shown in Figure 9 and Figure 10, FIRM provides the best average system performance and fairness of all studied scheduling schemes regardless of the BA-NVM latency. We conclude that FIRM can be effective with a wide variety of BA-NVM technologies and design points.

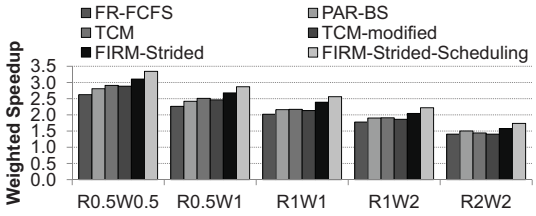

Fig. 9. System performance with various BA-NVM latencies.

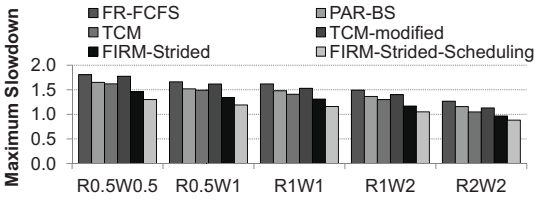

Fig. 10. Fairness with various BA-NVM latencies.

7.4. Sensitivity to BA-NVM Row-buffer Size

BA-NVMs can adopt various row-buffer sizes [45, 58]. We perform a sensitivity study with 1KB, 2KB (our default), 4KB, and 8KB row-buffer sizes. With each row-buffer size, we tune the granularity of persistent updates (the size of one log entry or one shadow update) to be $0.5\times$, $1\times$, $1.5\times$, and $2\times$ of row-buffer size as shown in the x-axis of Figure 11. We investigated persistent applications implemented with redo logging and shadow updates, respectively. Logging-based persistent applications allocate log entries in the contiguous physical address space. Therefore, persistent writes fill the BA-NVM row by row, regardless of the variation of value/element size. As a result, we observe that FIRM yields similar results with various row-buffer sizes for workloads that contain logging-based persistent applications (shown in Figure 11). The large performance and fairness improvements of FIRM are consistent, regardless of the row buffer size for such workloads.

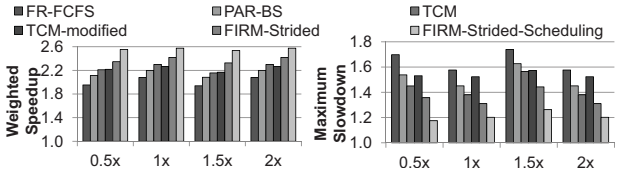

Fig. 11. Effect of BA-NVM row-buffer size.

FIRM's benefits with shadow update based persistent applications (not shown) depend on the relationship between the size of a persistent update and the row-buffer size. When the size of a persistent update is larger than the row-buffer size, the persistent update will fill the row buffer. Therefore, we observe similar results in such workloads as with logging-based persistent applications. We conclude that the benefits of FIRM are robust to variation in row buffer sizes, assuming persistent applications fully utilize the full row buffer during persistent updates.

7.5. Scalability with Number of Threads

Figure 12 demonstrates that FIRM outperforms all evaluated memory scheduling schemes on both system performance and fairness, when the number of threads varies from two to 16 (by employing one to eight processor cores).[14] In fact, FIRM benefits increase with more threads, due to increased interference between persistent and non-persistent applications. FIRM effectively reduces such interference. We conclude that FIRM benefits are robust to thread count.

[14]These average results are generated using multithreaded workloads, i.e., various persistent applications combined with *W4*, *W5*, and *W6*. Each persistent application and each of W4, W5, W6 executes the same number of threads as the number of cores when thread count changes.

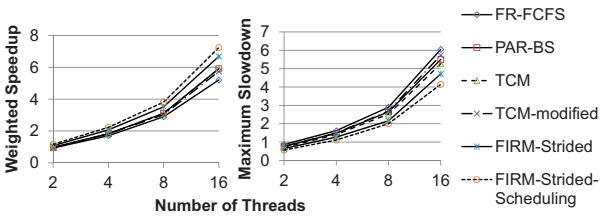

Fig. 12. Effect of different number of threads.

8. Discussion

BA-NVM Endurance. Among various BA-NVM technologies, STT-MRAM has near-SRAM endurance ($> 10^{15}$) [25, 44]. Therefore, endurance is not an issue in our implementation. Furthermore, FIRM does not apply any memory relocation in BA-NVM. *Persistent write striding* remaps memory accesses within the range of the physical address space allocated by persistent applications. Consequently, FIRM can be integrated with address-relocation based wear leveling mechanisms [72, 79] to address endurance issues, if the system adopts BA-NVMs with worse endurance, e.g., PCM (10^5-10^9) [5] and ReRAM (10^5-10^{11}) [40, 51].

Use Cases. Many scenarios are possible in general-purpose computing where persistent and non-persistent applications run concurrently. We provide two examples. First, in a multithreaded memory-hungry web browser application, BA-NVM can accommodate two types of threads concurrently: a front-end browser tab that uses the BA-NVM as working memory; a backend browser thread that caches user data to a persistent database [38]. Second, such mixing of applications can exist in consolidated cloud computing workloads, where various applications are packed in a limited number of servers. For example, one server can concurrently execute 1) a text search across terabyte-scale documents using in-memory indices and 2) a backend in-memory persistent file system. Many other examples abound in various domains of computing.

9. Related Work

Hardware resource contention in persistent memory systems received attention only in very recent works, which studied shared processor caches [38] and shared memory interface [52].

Kannan *et al.* observed that concurrently-running persistent and non-persistent applications tend to compete for the shared processor caches due to the doubled write requests issued by persistent applications to maintain versioning [38]. The study proposed a cache partitioning mechanism to address this issue.[15] Since FIRM addresses the contention at the memory interface, it can be combined with the mechanisms from [38]. Since all persistent writes need to go through the shared memory interface at some point, contention in shared memory can be a more common case in future memory-bandwidth-limited persistent memory systems.

Liu *et al.* studied the contention between persistent writes and non-persistent writes at the shared memory interface [52]. This study considers mainly writes and therefore does not take a comprehensive view of all memory scheduling problems in a system with persistent and non-persistent memory accesses. Because conventional memory scheduling policies deprioritize writes, manipulating the schedule of only writes may not effectively improve system performance (as we show in our evaluation of NVMDuet in Section 7.1). In contrast, we show that the contention between persistent writes and the traditionally-prioritized reads is critically important, leading to significant performance loss and unfairness with existing scheduling policies that do not handle this contention well. FIRM, which comprehensively considers the many problems caused by persistent application requests, therefore, provides higher system performance and fairness than NVMDuet [52], as shown in Section 7.1.

[15] Contention in shared caches may not exist in all persistent memory systems, however. Some persistent memory designs update data logs with uncacheable writes [90].

10. Conclusions

Emerging byte-addressable nonvolatile memory technologies open up promising opportunities for the creation of *persistent applications* that can directly and quickly manipulate persistent data via load and store instructions. This paper showed that when such applications contend with traditional non-persistent applications at the memory interface, existing memory scheduler designs lead to low system performance and low fairness across applications, since they cannot efficiently handle the intensive streaming write traffic commonly caused by persistent memory updates. We devised FIRM, which solves this problem with two novel key mechanisms: 1) striding of persistent memory writes such that they more effectively utilize memory parallelism and thus become less disruptive, 2) intelligent balanced scheduling of read and write requests of different types such that memory bandwidth loss is minimized and system performance and fairness are maximized. Our experimental evaluations across a variety of workloads and systems show that FIRM provides significantly higher system performance and fairness than five previous memory schedulers. We conclude that FIRM can provide an effective substrate to efficiently support applications that manipulate persistent memory in general-purpose systems. We also hope that the problems discovered in this paper can inspire other new solutions to shared resource management in systems employing persistent memory.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. Zhao and Xie were supported in part by NSF grants 1218867, 1213052, 1409798, and Department of Energy under Award Number DESC0005026. Mutlu was supported in part by NSF grants 0953246, 1065112, 1212962, 1320531, SRC, and the Intel Science and Technology Center for Cloud Computing.

References

[1] "Intel, persistent memory file system," https://github.com/linux-pmfs/pmfs.
[2] "Intel, a collection of linux persistent memory programming examples," https://github.com/pmem/linux-examples.
[3] "Intel Core i7," http://www.intel.com/content/www/us/en/processors/core/core-i7-processor.html.
[4] J. H. Ahn *et al.*, "McSimA+: a manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *ISPASS*, 2013.
[5] S. Ahn *et al.*, "Highly manufacturable high density phase change memory of 64Mb and beyond," in *IEDM*, 2004.
[6] R. Arpaci-Dusseau *et al.*, *Operating Systems: Three Easy Pieces*, 2014.
[7] R. Ausavarungnirun *et al.*, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.
[8] A. Badam, "How persistent memory will change software systems," *IEEE Computer*, 2013.
[9] K. Bailey *et al.*, "Operating system implications of fast, cheap, non-volatile memory," in *HotOS*, 2011.
[10] P. v. Behren, "SNIA NVM programming model," in *SNIA Education*, 2013.
[11] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[12] T. Bingmann, "STX B+ Tree, 2008," http://panthema.net/2007/stx-btree.
[13] T. C. Bressoud *et al.*, "The design and use of persistent memory on the DNCP hardware fault-tolerant platform," in *DSN*, 2001.
[14] C. Cagli, "Characterization and modelling of electrode impact in HfO2-based RRAM," in *Workshop on Innovative Memory Technologies*, 2012.
[15] Y. Choi *et al.*, "A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth," in *ISSCC*, 2012.
[16] J. Coburn *et al.*, "NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *ASPLOS*, 2011.

[17] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.
[18] G. Copeland *et al.*, "The case for safe RAM," in *VLDB*, 1989.
[19] R. Das *et al.*, "Application-to-Core Mapping Policies to Reduce Memory System Interference in Multi-Core Systems," in *HPCA*, 2013.
[20] R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.
[21] R. Degraeve *et al.*, "Dynamic hourglass model for SET and RESET in HfO2 RRAM," in *VLSI*, 2012.
[22] G. Dhiman *et al.*, "PDRAM: A hybrid PRAM and DRAM main memory system," in *DAC*, 2009.
[23] X. Dong *et al.*, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *SC*, 2010.
[24] X. Dong *et al.*, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, 2012.
[25] A. Driskill-Smith, "Latest advances in STT-RAM," in *NVMW*, 2011.
[26] E. Ebrahimi *et al.*, "Prefetch-aware shared resource management for multi-core systems," in *ISCA*, 2011.
[27] E. Ebrahimi *et al.*, "Parallel application memory scheduling," in *MICRO*, 2011.
[28] F. Eskesen *et al.*, "Software exploitation of a fault-tolerant computer with a large memory," in *FTCS*, 1998.
[29] S. Eyerman *et al.*, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[30] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs," *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2011.
[31] M. Hosomi *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-RAM," in *IEDM*, 2005.
[32] Intel Corporation, "Intel architecture instruction set extensions programming reference, 319433-012 edition," 2012.
[33] E. Ipek *et al.*, "Dynamically replicated memory: Building reliable systems from nanoscale resistive memories," in *ASPLOS*, 2010.
[34] E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
[35] C. Jacobi *et al.*, "Transactional memory architecture and implementation for IBM System Z," in *MICRO*, 2012.
[36] J. Janesky, "Device performance in a fully functional 800MHz DDR3 Spin Torque Magnetic Random Access Memory," in *IMW*, 2013.
[37] J.-Y. Jung *et al.*, "Memorage: Emerging persistent RAM based malleable main memory and storage architecture," in *ICS*, 2013.
[38] S. Kannan *et al.*, "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *HPCA*, 2014.
[39] W. Kim *et al.*, "Extended scalability of perpendicular STT-MRAM towards sub-20nm MTJ node," in *IEDM*, 2011.
[40] Y.-B. Kim *et al.*, "Bi-layered RRAM with unlimited endurance and extremely uniform switching," in *VLSI*, 2011.
[41] Y. Kim *et al.*, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.
[42] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
[43] Y. Kim *et al.*, "A case for exploiting subarray-level parallelism (SALP) in DRAM," in *ISCA*, 2012.
[44] E. Kultursay *et al.*, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
[45] B. C. Lee *et al.*, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.
[46] B. C. Lee *et al.*, "Phase change memory architecture and the quest for scalability," *CACM*, 2010.
[47] B. C. Lee *et al.*, "Phase-change technology and the future of main memory," *IEEE Micro*, 2010.
[48] C. J. Lee *et al.*, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.
[49] C. J. Lee *et al.*, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," *HPS Tech. Report*, 2010.
[50] C. J. Lee *et al.*, "Improving memory bank-level parallelism in the presence of prefetching," in *MICRO*, 2009.
[51] W. S. Lin *et al.*, "Evidence and solution of over-RESET problem for HfOx based resistive memory with sub-ns switching speed and high endurance," in *IEDM*, 2010.
[52] R.-S. Liu *et al.*, "NVM Duet: Unified working memory and persistent store architecture," in *ASPLOS*, 2014.
[53] Y. Lu *et al.*, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *ICCD*, 2013.
[54] Y. Lu *et al.*, "Loose-ordering consistency for persistent memory," in *ICCD*, 2014.
[55] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[56] P. E. McKenney, "Memory barriers: a hardware view for software hackers," 2009, http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf.
[57] J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE CAL*, 2012.
[58] J. Meza *et al.*, "A case for small row buffers in non-volatile main memories," in *ICCD*, 2012.
[59] J. Meza *et al.*, "A case for efficient hardware/software cooperative management of storage and memory," in *WEED*, 2013.
[60] I. Moraru *et al.*, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *TRIOS*, 2013.
[61] T. Moscibroda *et al.*, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.
[62] T. Moscibroda *et al.*, "Distributed Order Scheduling and its Application to Multi-Core DRAM Controllers," in *PODC*, 2008.
[63] S. P. Muralidhara *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO*, 2011.
[64] O. Mutlu *et al.*, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.
[65] O. Mutlu *et al.*, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.
[66] D. Narayanan *et al.*, "Whole-system persistence," in *ASPLOS*, 2012.
[67] K. J. Nesbit *et al.*, "Fair queuing memory systems," in *MICRO*, 2006.
[68] W. Otsuka *et al.*, "A 4Mb conductive-bridge resistive memory with 2.3GB/s read-throughput and 216MB/s program-throughput," in *ISSCC*, 2011.
[69] H. Park *et al.*, "Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems," in *ASPLOS*, 2013.
[70] S. Pelley *et al.*, "Memory persistency," in *ISCA*, 2014.
[71] S. Pelley *et al.*, "Storage management in the NVRAM era," *VLDB*, 2013.
[72] M. K. Qureshi *et al.*, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *MICRO*, 2009.
[73] M. K. Qureshi *et al.*, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.
[74] L. E. Ramos *et al.*, "Page placement in hybrid memory systems," in *ICS*, 2011.
[75] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM JRD*, 2008.
[76] S. Rixner, "Memory controller optimizations for web servers," in *MICRO*, 2004.
[77] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.
[78] V. Seshadri *et al.*, "The dirty-block index," in *ISCA*, 2014.
[79] A. Seznec, "A phase change memory as a secure main memory," *IEEE CAL*, 2010.
[80] A. Snavely *et al.*, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS*, 2000.
[81] V. Sousa, "Phase change materials engineering for RESET current reduction," in *Workshop on Innovative Memory Technologies*, 2012.
[82] SPEC CPU, "SPEC CPU2006," http://www.spec.org/cpu2006/.
[83] J. Stuecheli *et al.*, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *ISCA*, 2010.
[84] L. Subramanian *et al.*, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.
[85] L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.
[86] R. Takemura *et al.*, "A 32-Mb SPRAM with 2T1R memory cell, localized bi-directional write driver and '1'/'0' dual-array equalized reference scheme," *JSSC*, 2010.
[87] H. Vandierendonck *et al.*, "Fairness metrics for multi-threaded processors," *IEEE CAL*, 2011.
[88] S. Venkataraman *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory," in *FAST*, 2011.
[89] Viking Technology, "Understanding non-volatile memory technology whitepaper," 2012, http://www.vikingtechnology.com/uploads/nv_whitepaper.pdf.
[90] H. Volos *et al.*, "Mnemosyne: lightweight persistent memory," in *ASPLOS*, 2011.
[91] H. Yoon *et al.*, "Row buffer locality aware caching policies for hybrid memories," in *ICCD*, 2012.
[92] J. Zhao *et al.*, "Kiln: Closing the performance gap between systems with and without persistence support," in *MICRO*, 2013.
[93] W. Zhao *et al.*, "Macro-model of spin-transfer torque based magnetic tunnel junction device for hybrid magnetic-CMOS design," in *BMAS*, 2006.
[94] P. Zhou *et al.*, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009.
[95] W. K. Zuravleff *et al.*, "Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order," *U.S. Patent Number 5,630,096*, 1997.